Visualizing Message Patterns in Object-Oriented Program Executions

Dean F. Jerding[1], John T. Stasko[1], and Thomas Ball[2]

[1]Graphics, Visualization, and Usability Center
Georgia Institute of Technology
Atlanta, GA 30332-0280
{dfj,stasko}@cc.gatech.edu

[2]Software Production Research
Bell Laboratories
Naperville, IL 60566
tball@research.att.com

Technical Report GIT-GVU-96-15
May 1996

Abstract

The dynamic behavior of object-oriented programs is difficult to design, implement, and modify. Understanding the interactions between classes and objects is necessary to create efficient designs and make safe modifications. This work seeks to identify, visualize, and analyze recurring message patterns in object-oriented program executions as a means for understanding and examining dynamic behavior. Our visualizations focus on supporting design recovery, validation, and reengineering tasks.

Keywords: software visualization, object-oriented programs, program understanding, software reengineering

1 Addressing Problems of OO Dynamics

The importance of dynamic behavior in the design and implementation of object-oriented (OO) systems cannot be over-emphasized. Object-oriented analysis and design techniques (OOA/D) that evolve around object models created from static problem statements or object decomposition of real-world systems must not avoid the dynamic issues. Usage-scenario based approaches to OOA/D use the dynamics of the system to help design the object models. Still, object models emphasize (and sometimes over-use) inheritance because languages like `C++` and `Smalltalk` formally support them. However, the complexity and beauty of designs lie in the class and object associations through which objects interact to accomplish tasks. Not only is it difficult to design these dynamic relationships, standard languages do not provide implementation support for interactions as first-class entities.[1]

The communication dialog between classes and objects is typically designed using graphical notations such as event trace diagrams or interaction diagrams[RBP+91, CAB+94]. Depending on the size and complexity of the system being developed, these design documents may be used sparingly or often. Under-utilization of these development techniques may occur because it is difficult to map these diagrams into the implementation.

We use graphical visualizations to display and examine the dynamic behavior of object-oriented programs. Interactive graphical visualizations can present this voluminous information much more effectively than textual representations, allowing a user to control the filtering and abstraction of available information. We have created scalable visualizations to examine real-world sized message traces. Using these views we have observed definite interaction patterns in OO systems.

Interaction patterns are manifested as repeated sequences of messages and recurring instantiation of objects. *Message patterns* will occur when similar semantic operations are performed or during iteration as in a loop. They can result from a program's design, architecture, implementation, or usage. This paper discusses our efforts to identify, analyze, and understand these patterns.[2]

Our visualization prototypes can automatically identify message patterns and then allow users to examine and manipulate them. In order to store and analyze large message traces, a compact representation of the call trace has been developed. We hypothesize that high-level (design level) program behavior can be abstracted out from the low-level message trace via the message patterns. Visualizations of the abstract behavior can then be compared with design level information, such as execution scenarios or interaction diagrams, to help extract and validate the design and implementation. An example scenario showing how this can be done is described in the next section.

The aim of our visualizations is to facilitate design recovery, validation, and reengineering tasks by exposing dynamic interactions. Section 2 describes visualizations we have created, how message patterns are identified, and an example usage scenario. The third section describes related work, and the last section discusses the impact and future work.

[1] To meet this need, several research efforts have investigated language support for associations[Kri94, DBFPD95].

[2] Note that our use of the word "pattern" is different than that of *design patterns* or *pattern languages* [AIS+77, GHJV95, CS95], yet the two are related. See Section 3 on Related Work.

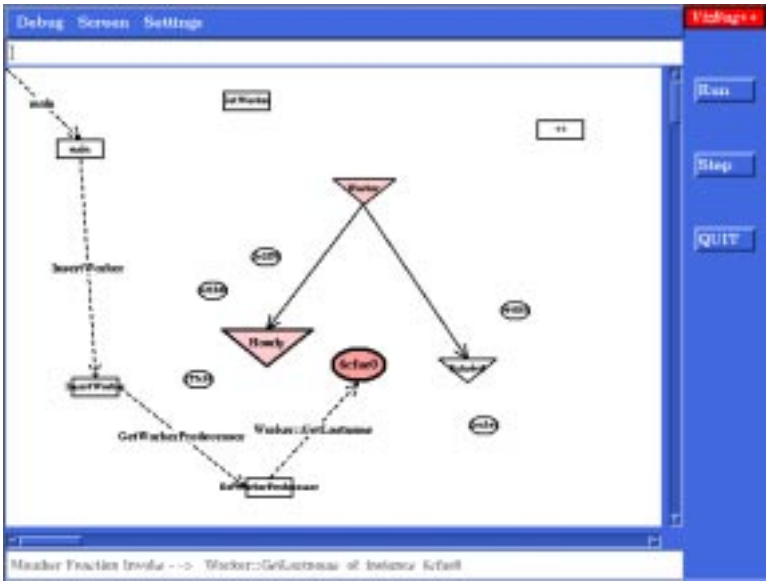


Figure 1: Prototype visualization of simple C++ worker database program. Ovals represent instances, inverted triangles are classes, and rectangles global functions. Message passing is animated by drawing arrows between objects and functions, and instances "hatch" out of their classes.

2 Visualizing Message Patterns

Our work on visualizing the dynamics of object-oriented systems began with the idea that it might be useful to expose the interaction between objects as an OO system executes. As designers or programmers, we have a mental model of how we expect the program to operate (often based on the design), yet many factors can cause a mismatch between this and the actual behavior.

The first visualizations we developed animated the instantiation of objects and the message passing between objects in a single view, as in Figure 1. The visualization was based on trace files generated from `C++` source code annotated by hand. Only small, toy programs were visualized. The next generation of visualization used multiple views to show the call stack, inheritance hierarchy, instances, and message passing[JS94]. Animation was still used to show the progression of time, but larger programs were visualized.

2.1 The Execution Mural

After working with these early prototypes, we realized that animation is less important for global program understanding tasks than being able to browse the entire execution and focus on areas of interest as needed to undercover specifics. Thus, we chose to focus on views that showed the progression of time inherently as one of the dimensions. We also narrowed our focus to the sequence of object interactions, while others published work focused on cumulative dynamics[DPHKV93, DPKV94]. Scalability became the number one issue in creating useful visualizations to aid the understanding processes during implementation and maintenance of real-world sized systems.

The **Execution Mural** view of Figures 2 - 3 shows a class level event trace diagram of a particular program execution. We take the approach of creating a general view which initially shows all the classes and messages, and providing several visual filtering mechanisms which allow a

user to focus on information of interest. These include the ability to change the ordering of classes along the vertical axis, selectively show or hide particular classes, color-code specific messages, and zoom in on sub-sections of the message trace. Note that both function invocations and function returns are treated as messages.

Sample Execution Murals from a Polka[SK93][3] bubble-sort algorithm animation are shown in Figures 2 and 3. The information used to create these views was obtained from a static analysis of the source code using `gen++`[Dev92] and a trace file produced from source which was automatically annotated by a `Perl` script which places tracing objects in the code as described by O'Riordan[O'R88].

The major visual innovation in the Execution Mural is the ability to create a global overview of a message trace containing hundreds of thousands of messages. The technique utilizes grayscale and color shading along with anti-aliasing techniques to create a miniature representation of an entire large information space. Such a view is called an *Information Mural*; the technique is described in [JS95a, JS95b].

What these message "murals" have allowed us to do is notice visual patterns in entire message traces, and then lower-level patterns as we zoom in on sub-sequences of the execution. After using the Execution Mural view to examine executions, it became obvious that the visual patterns are either the result of similar semantic operations in the code or of iteration. One of the weaknesses of the visualization in terms of helping program understanding tasks is that the view of individual messages is too low-level compared to a user's mental model.

2.2 Identifying Message Patterns

We see repeated message sequences, or *message patterns*, as abstractions that correspond to higher level operations in the code which are evident in design models. The natural next step for us was to identify these message patterns automatically and treat them as first-class entities in our visualization. Message patterns could then act as a starting point for design validation and reengineering tasks, allowing the user's understanding of these patterns to facilitate a comparison of expected and observed behavior and help uncover areas which need to be reengineered. However, in order to visualize and analyze large program executions, a compact representation of the message trace and a way to extract the occurrences of message patterns is required.

In a spectrum of possible representations of calling behavior that pit space overhead versus information accuracy, the call graph and the dynamic call trace represent two extreme endpoints. We have developed a middle ground that allows a range of possibilities in this tradeoff. Our data structure also provides various abstract views of the dynamic information and serves well as a query engine for software tools dealing with calling behavior. One such abstract view of this data structure is the notion of a message pattern.

At one extreme, a call graph is a compact representation of calling behavior that summarizes all possible run-time activation stacks. The call graph contains one vertex for each procedure in a program and an edge from A to B if A calls B.[4]

There is much interesting information about calling behavior that is dropped to gain compactness. The sequencing of calls, the context in which certain calls are made, conditional and indirect

[3] Polka is an object-oriented toolkit for creating algorithm and program visualizations, written in `C++`.

[4] Of course, in the presence of indirect calls, the problem of determining the target of the call is generally undecidable.

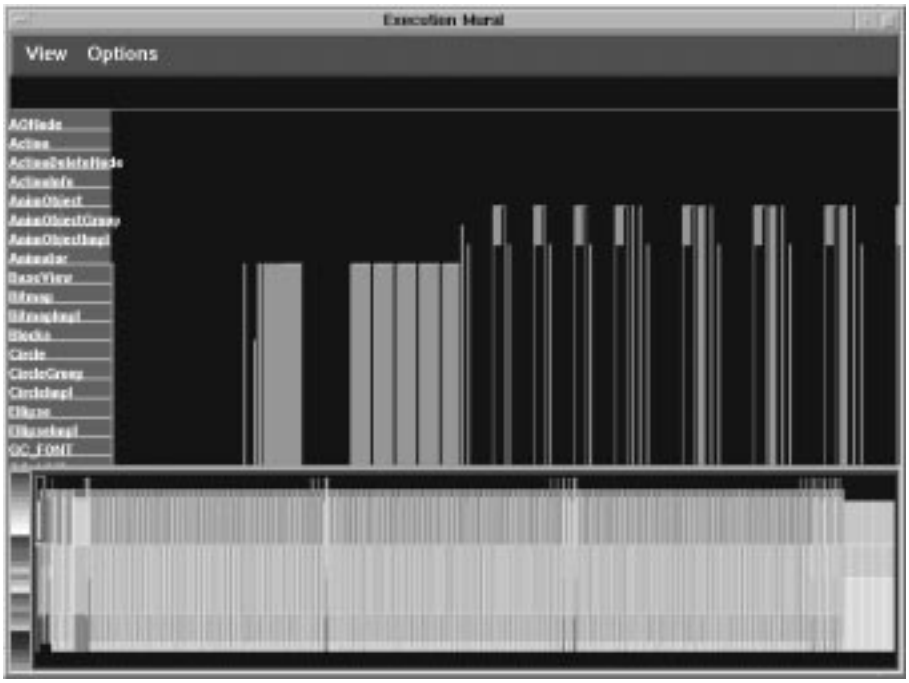


Figure 2: Initial Execution Mural of a Polka bubble-sort. Classes are on the vertical axis, and each message is single pixel wide line from source to destination class. The global view at the bottom is a mural of the entire set of over 52,000 messages; a navigation rectangle shows the position of the focus area relative to the rest of the classes and messages in the trace.

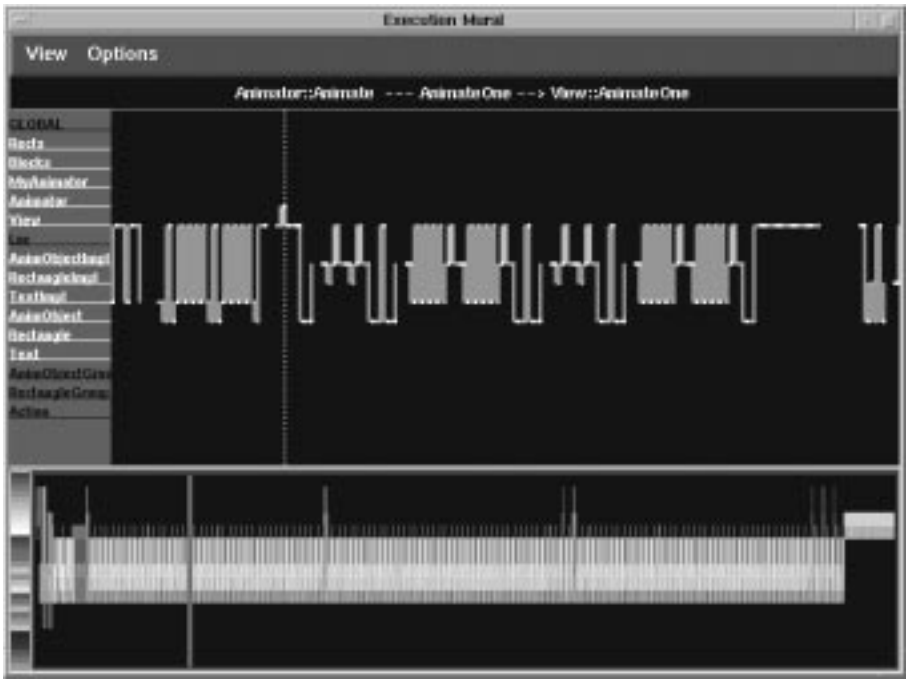


Figure 3: Execution Mural from Figure 2 with several classes hidden, the message width increased, and several messages highlighted in different colors.

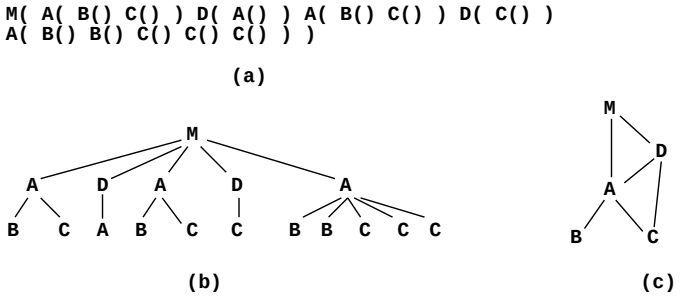


Figure 4: (a) A call trace, (b) its corresponding call tree, and (c) call graph. Edge directions are assumed to be directed down the page.

calls, and repeated calls are all examples of calling behavior that are lost. Such problems manifest themselves in a number of software tools that use the call graph to summarize dynamic program properties. For example, the inaccuracy of program profilers such as gprof [GKM83] and qpt [LB92] can be traced to their use of the call graph to summarize context-dependent profile information in a context-independent manner.

At the other end of the spectrum, the dynamic call trace is an unbounded data structure containing a record of all the calls and returns that occur in a program's execution, regardless of whether the calls are direct or indirect. Extracting the call trace may incur high run-time overhead and storing the trace may not be feasible for long running programs[Moh88]. Furthermore, there is a data explosion problem: finding interesting information from the mass of data in the trace is not easy. Some trace-based tools animate the call graph to show the trace on the fly (without storing it) [BH90], or compute statistical summary information from the trace [DPKV94]. Both of these techniques deal with the space problem by ignoring or summarizing a large amount of dynamic information, as is done with the call graph.

We would like to have the best of both worlds: a compact representation (such as the call graph) that also retains as much information as possible about dynamic calling behavior (such as the dynamic call trace). We compactly represent the dynamic call trace of a program's execution using a simple analysis and data structure.

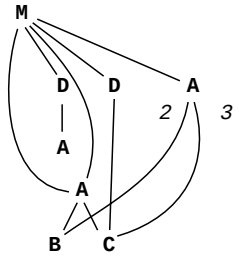


Figure 5: A compact representation of the call tree from Figure 4b.

There are three basic ideas we use to compact the dynamic call tree. First, we use hash consing to ensure that the tree structures derived from the dynamic call trace are represented exactly once in the compact representation. Second, we compactly summarize repetitive sequences of subtrees that are generated by loops. These sequences can be summarized at varying degrees of accuracy, resulting in different compact representations (and subsequently different levels of message pattern abstractions). Finally, we compress repetitive calling chains that are generated by recursion. The compact representation of the dynamic call tree is a directed acyclic graph (`dag`).

Figure 5 shows one possible compact representation of the call tree from Figure 4(b). Each vertex corresponds to a call. It is clear that this representation captures exactly the same information as the call tree.

The basic framework for parsing a call trace to produce a `dag` is straightforward. The analysis

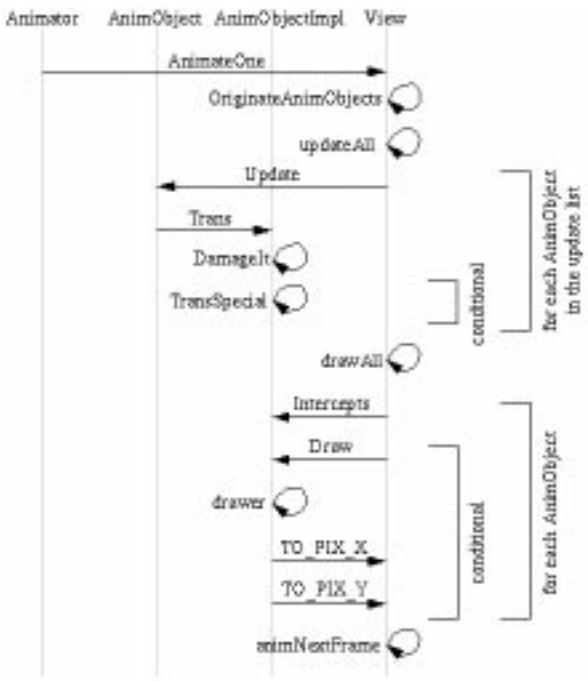


Figure 6: Event trace diagram of process to animate one frame in the Polka animation toolkit.

requires three main data structures: a stack of active procedures, a heap of `dag` structures, and a hash table for determining if a particular `dag` structure has been built already. The `dag` structures are built in a bottom-up fashion (from the leaves of the dynamic call tree to the root). Hash consing ensures that if a tree data structure is constructed bottom-up, then duplicate trees will always hash to the same element.

Hash consing results in the sharing of subtrees in the `dag`, as is evident by the shared subtree of A calling B and C in Figure 5. This subtree is a message pattern, because it has more than one incoming parent edge. We have implemented a pattern iterator that walks the `dag` and returns message patterns that are encountered. In addition to looking for shared subtrees, the pattern iterator also looks for repetitive subtrees sequences that resulted from iteration in the program execution.

2.3 Message Pattern Views

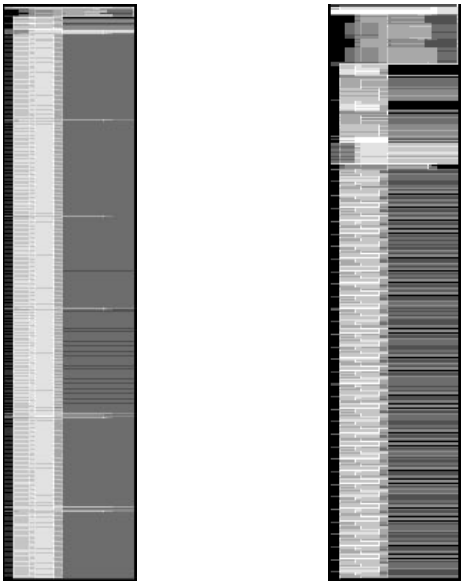
Our current visualization prototype is focused on visualizing interaction patterns to support design recovery and reengineering tasks. The automatic message pattern detection described in the last section is used as a starting point for presenting the user with existing patterns. The visual interface then allows the user to examine the message patterns and look for new ones at various levels of abstraction. In this section we describe our prototype and report results of using several different message pattern-oriented views.

The prototype allows us to create views of a particular program execution based on static information about the program and a trace file of interesting events (function calls and returns). The views are *Observers*[GHJV95] of a single program model which contains both static and dynamic information, and they co-exist in a single **Viewspace** window which acts as a *Controller*[Gol83] to handle user input. A *Composite*[GHJV95] class hierarchy defines views as visual objects themselves.

Interaction occurs through pointing and using pop-up menus which are associated with the various views.

Usage Scenario. Here we use our tool to examine the Polka program animation toolkit mentioned in Section 2.1. We are interested in comparing the Polka toolkit designer's mental model of its behavior with the actual implementation. The focus is on the interactions taking place as each animation frame is rendered. Figure 6 shows an event trace diagram made by the Polka designer to describe the interactions involved in Polka while animating a frame.

A trace file for a Polka bubblesort algorithm animation which consists of almost 64,000 function invocations is read and processed by our system. We first create a global **Execution Mural** (Figure 7a) of the entire message trace. This view will act as a global overview, showing where the message patterns that are identified fit within the execution. Notice that the Execution Mural views in this prototype are different from the previous generation described in Section 2.1; they have been rotated to look more like interaction diagrams, with the 40 classes in the program on the horizontal axis and the almost 64,000 messages drawn as horizontal lines down the vertical axis using the Information Mural compression technique[JS95a]. Areas that are brighter in the mural are more dense with information, conveying the same visual patterns that would be apparent if a huge event trace diagram of the entire program was observed from a distance. The global Execution Mural does not have a focus area, it just shows all of the messages at once.

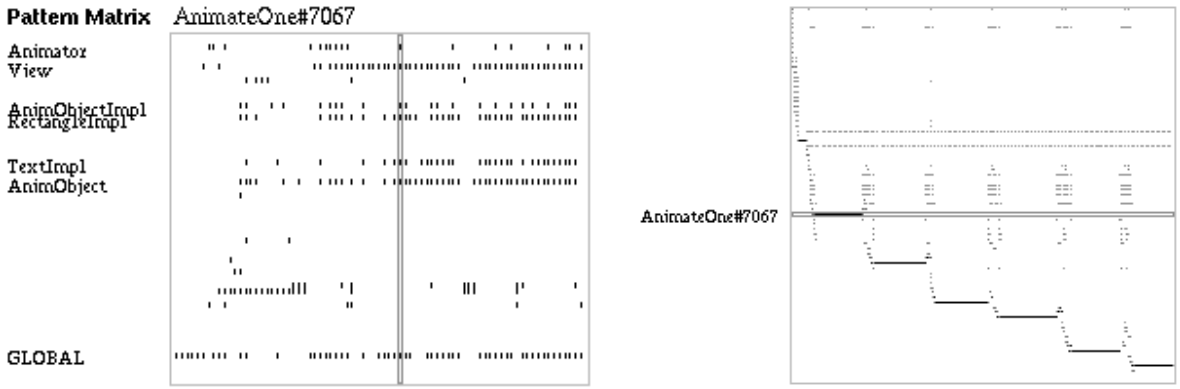


(a) (b)

Figure 7: (a) Global Execution Mural for the Polka bubblesort animation, essentially a miniature event trace diagram of the entire message trace. The vertical resolution is 64,000 messages on 400 pixels, an information compression ratio of 160:1. (b) Part (a) zoomed in on approximately the first 10,000 messages of the trace. The information compression ratio is around 25:1.

Notice how repetitive the diagram is visually. A distinct pattern appears in the beginning, followed by another that repeats six times. To get a feel for the information compression ratio, Figure 7b shows approximately the first 10,000 function invocations in the trace. In part (b) more visual patterns become apparent as we zoom in to a finer resolution.

Now we create a **Pattern Matrix** (Figure 8a) showing the classes involved in the top-level message patterns that were identified by our system. "Top-level" means the largest sequence of messages that occur more than once and begin closer to the root of the call graph than other sub-sequences that might also be message patterns. The matrix assigns a message pattern to each column; message patterns are identified by the first message name along with the global message



(a) (b)

Figure 8: (a) Pattern Matrix for the Polka bubble-sort animation. Message patterns are assigned columns in the matrix, and rows are classes in the program. Entries are made for classes which are "members" of each message pattern. Pattern AnimateOne#7067 is currently highlighted. (b) Pattern Mural for the Polka bubble-sort animation. Message patterns are assigned to the vertical axis by order of occurrence; a point in the mural is made for each message at sequential x coordinates, with the y value corresponding to the pattern of which that message is a member (messages that are not members of patterns are not shown). The AnimateOne#7067 message pattern is highlighted.

number of the first message in the pattern. The rows of the matrix correspond to the classes in the program. The matrix is created using the Information Mural technique as well, so is effectively scalable to hundreds of classes and patterns. Note that the order of message patterns along the horizontal axis can be changed to group patterns by name, size, or order of occurrence in the trace.

From this view we can see which patterns might be related by looking for ones which contain the same classes. The designer's diagram (Figure 6) includes the `Animator`, `View`, `AnimObject`, and `AnimObjectImpl` classes. The Pattern Matrix has several `AnimateOne` patterns which contain most of these classes, for example Figure 8a has the `AnimateOne#7067` pattern highlighted, which contains classes `Animator`, `View`, `AnimObjectImpl`, `RectangleImpl`, `TextImpl`, `AnimObject`, and the `GLOBAL` class which represents functions in the global scope. This message pattern is also a likely candidate because its first message is `AnimateOne`, which is same as the first one in the designer's diagram.

The **Pattern Mural** gives a time ordering to the message patterns shown in the matrix by showing message patterns on the vertical axis, and where they occur in the program execution along the horizontal. This view uses the Information Mural technique by drawing a point for each message in the execution, at sequential x coordinates and at the appropriate y coordinate for the message pattern to which that message belongs. Note that "sequential x coordinates" are in terms of the message order, not the pixels on the screen: many messages may be compacted into the same column of pixels.

The order of patterns along the vertical axis can be changed as in the Pattern Matrix view; Figure 8b shows patterns in order of occurrence (first at the top). In this view we notice several distinct `AnimateOne` patterns which occur in the middle of the trace. We hypothesize that each of these patterns corresponds to the distinct phases in the global Execution Mural of Figure 7a. If

we turn on highlighting of the current selected pattern in the global Execution Mural, we confirm this suspicion. Figure 9 shows the location of `AnimateOne#7067`. Note that all the views are synchronized so that as we change the current pattern in one view the others change to show the location of that pattern as well.

We have been focusing on the `AnimateOne` pattern because it seems be the one we are looking for to compare with the designer's mental model of frame animation. We now use our system to create an Execution Mural of the `AnimateOne#7067` pattern, shown in Figure 10. The mural on the right hand side provides a global overview of all the messages in the pattern, and acts as a two-dimensional scroll bar for moving the focus area on the left. Currently messages corresponding to both function calls and returns are displayed; calls are solid and returns are grayed. Horizontal lines represent messages, and name labels can optionally be displayed above each message. A circle marks the destination end of the message.

Figure 9: Global Execution Mural with message pattern AnimateOne#7067 highlighted in red (the dark area near the top).

Because the designer's event trace diagram does not include global function calls (they are mostly for the graphics), we can remove the `GLOBAL` class by using the mouse to select the class label and choosing a menu option to remove that class. Another menu option allows us to eliminate the return messages from the display. The Execution Mural now appears as in Figure 11.

Figures 11 - 13 show different parts of the extracted message pattern. We can compare these with Figure 6 to see how the implementation conforms to the expected design.

From a quick glance through the entire pattern, we see good conformance to the expected behavior. There are four `Update`s, two which deal with `RectangleImpl`s and two which deal with `TextImpl`. The designer did not include these classes in his diagram, presumably because they are specializations of the `AnimObjectImpl` class. Because it is common for design models to deal with abstract classes, a feature we are adding to the Execution Mural will allow subclass behavior to be generalized into the base class.

The designer quickly concludes that in this frame of the bubble-sort animation two bars with their text labels are changing places. There are some messages (`BoundBox`, `DamageCheck`) which are not in the designer's diagram, but are in the correct place according to the designer. Note that the original diagram only had one `DamageIt` message after the `Trans` message, where the observed pattern has two. The designer confirms that there should be two, because one is for the old position of the object and one is for the new position after the object moves, changes size, or does some other action. Here is a case where the design model would need to be updated.

After the `drawAll` message shown in Figure 12, every `AnimObject` should be redrawn if it intersects the "damaged region." Again the `BoundBox` message has been omitted in the designer's diagram, and we can see one `Intercepts` message which is not followed by a `Draw` and one that is (a `RectangleImpl`). Looking at the pattern in the global mural, there are in fact two `RectangleImpl` that are drawn, as expected. Continuing to examine the rest of the `Intercept`s, there are two that result in `Draw`s to `TextImpl`s. We do see a number of messages in the pattern that are not in the diagram, such as `PIXX` and `PIXY` which are access functions. The `animNextFrame` message ends the

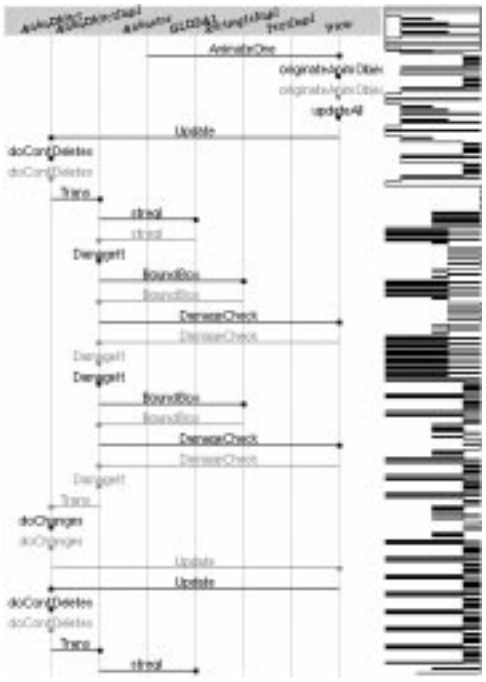


Figure 10: Execution Mural of message pattern AnimateOne#7067. The mural on the right gives a global view of all messages in the pattern and acts as a two-dimensional scrollbar for the focus area. The navigation rectangle in the upper part of the global view corresponds to the messages displayed in the focus area.

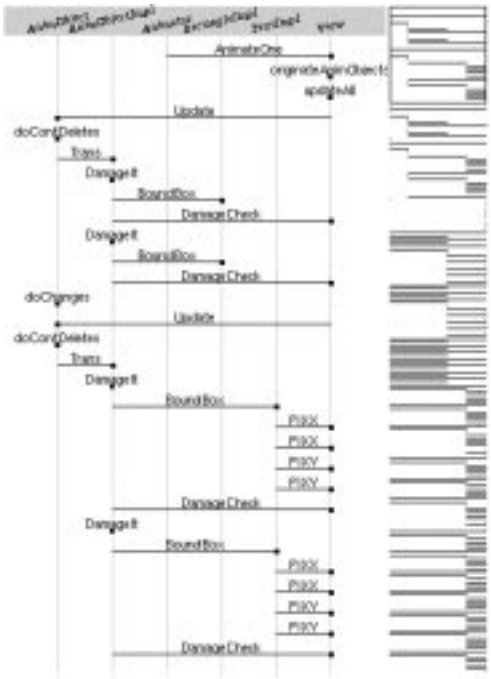


Figure 11: Execution Mural of message pattern AnimateOne#7067 with GLOBAL class removed and showing only messages corresponding to function calls.

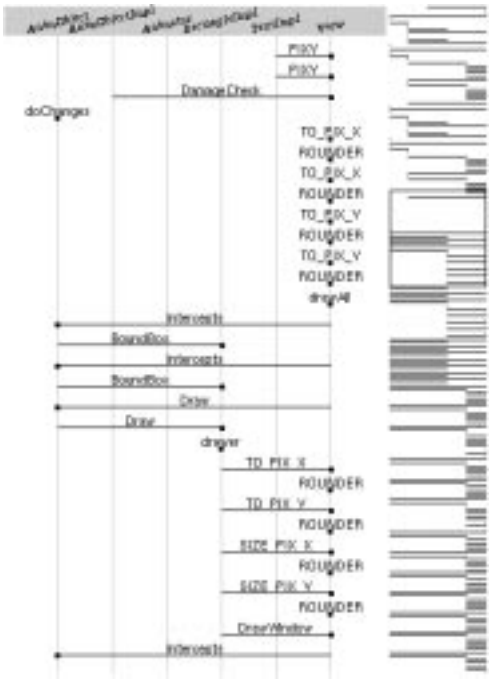


Figure 12: Execution Mural of message pattern AnimateOne#7067 from Figure 11 scrolled part way down.

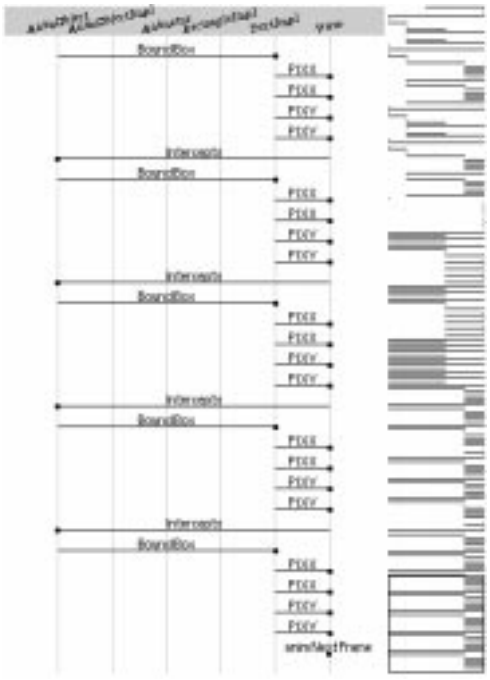


Figure 13: Execution Mural of message pattern AnimateOne#7067 from Figure 11 scrolled all the way down.

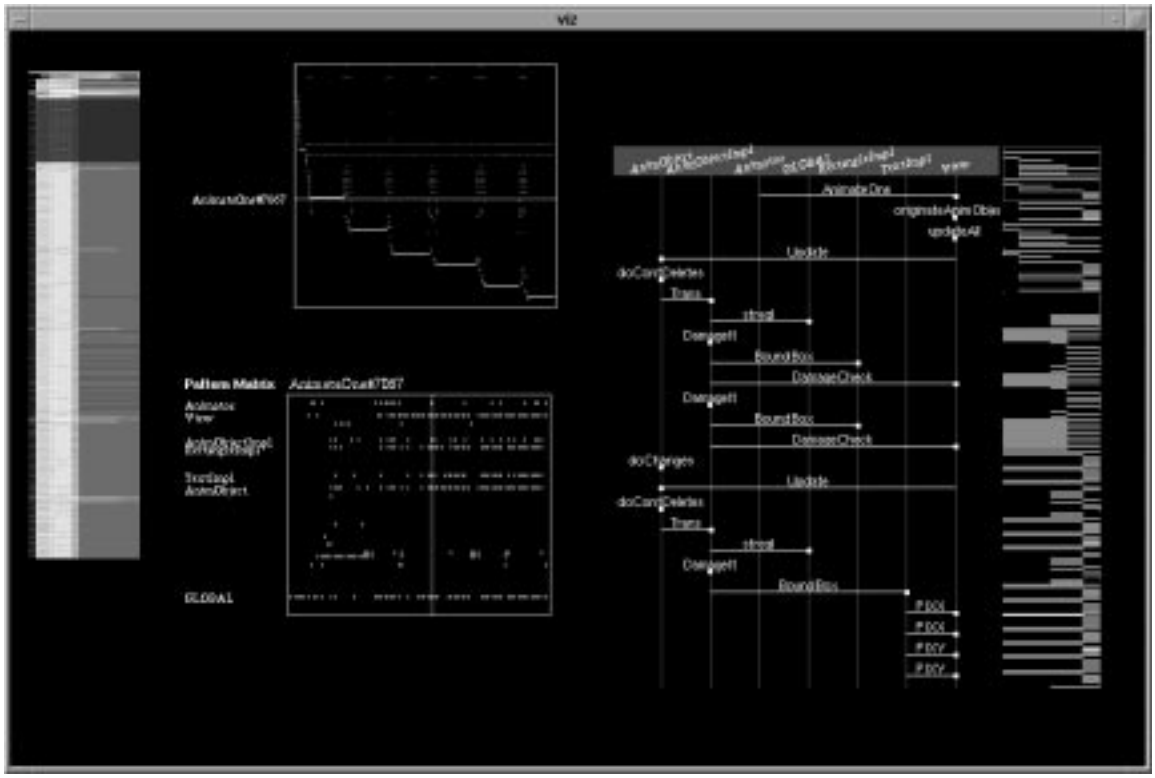


Figure 14: Snapshot of the viewspace containing the views described in Figures 7-13.

pattern as it does in the diagram.

So, for this example the `AnimateOne#7067` pattern does in fact appear to implement the frame animating process as expected. After investigating Execution Murals of other `AnimateOne` patterns, it seems that the differences between them result from `AnimObject`s first originating in particular frames or different `AnimObject`s being updated and drawn. Our visualizations should make it easy to uncover these differences. Figure 14 shows all of the views mentioned thus far together in the viewspace.

3 Related Work

Several different areas overlap with our work, including software visualization, program understanding, reverse engineering, and OO methods. Some of the more recent efforts in these areas are mentioned here and related to our work.

The notion of a pattern as a solution to a problem in a particular context provides a literary form through which experience with software can be documented to be reused by others[AIS+77, GHJV95, CS95]. Patterns in the software domain range across many levels of abstraction, from the organization, to the design, architecture, implementation, and programming language. Our message patterns are so named because they too are a repeatable entity and because they create visual "patterns" on the screen when we visualize OO message traces. The relation between the two arises in that message patterns will result from various design and implementation patterns, and can be seen as low-level evidence for the existence of a design pattern. In this way, identifying

message patterns can be seen as a form of "pattern mining."

The Program Explorer[LN95b, LN95a] is a `C++` program understanding tool that is focused on class and object centered views. The authors have developed a robust, class-wise tracing system for tracking function invocation, object instantiation, and attribute access. Static information in a program database is used to leverage the dynamic information from the tracing environment, and vice-versa, which makes sense in a program understanding environment. The views show class and instance relationships (usually focused on a particular instance or class), and short method invocation histories. It seems that the system is designed to execute the program for a while, and then focus on particular classes or objects. It's not intended as a global understanding tool, so the user must know what (or where in the execution) they are interested in beforehand. In [LN95a] examples of using the system to uncover design patterns in real-world sized systems are given. Again, though, it seems that the user must know the pattern and have an idea where that pattern occurs to exploit the visualizations.

HotWired is a visual debugger for `C++` and `Smalltalk` that provides both standard object views and a scripting language to create simple program visualizations[LM94]. Views show instances of classes (similar to [DPHKV93]), message passing between individual instances, and instance attribute values. It is possible to "record" particular message traces to be replayed. Their recording strip view shows instance activation over time, and could benefit from our Information Mural technique. The visualizations focus on debugging tasks, such as examining sequences that lead up to a particular error. To support custom debugging, a scripting language maps instance values to visual objects. Because these scripts must be manually written, this facility seems most useful for situations that recur often, such as explaining some program behavior. Debugging tasks are usually performed once or repeated until the problem is solved.

De Pauw, Helm, Kimelman, and Vlissides[DPHKV93, DPKV94] have developed visualization techniques and a tool for presenting attributes of object-oriented systems, more specifically, `C++` programs. The authors use portable instrumentation techniques to extract the required information about a program's execution. They also developed views, most of which are chart-like, that present summary information about the execution. The views display instance creation and destruction, inter- and intra-class calls, allocation histories, and so on. These views are quite effective for analyzing program performance and class relationships in terms of the amount of interaction between classes and objects. However, the information they capture is mostly post-mortem summary information, whereas we seek to uncover the semantics and sequence of the interactions. The authors made this compromise when they decided not to store incremental information about the execution in favor of storing more cumulative information. Thus, the actual message trace cannot be reconstructed based on their database.

The OO!CARE tool is the `C++` version of the CARE program understanding environment[LC94]. The idea of the OO!CARE system is to extract and visualize dependencies between classes, objects, and methods in the program, as well as the control and data flow. The system includes a code analyzer, a dependencies database, and a display manager. The hierarchically designed views present class inheritance, control-flow dependencies, and file dependencies. A column oriented view called a *collonade* presents data-flow dependencies. The dependencies are extracted statically, so in the case of a virtual function call in C++ a "dummy" member function is created to represent all the possible run-time bindings. While the views provide zooming and panning capabilities, plus hierarchical decomposition, the examples given do not demonstrate that they scale to handle large programs.

Murphy, et al. have developed an approach that allows software engineers specify high-level

models of a system and how the source code maps into that model[MNS95]. Then a *reflexion model* is computed which uses call graph and data referencing information to determine where the model agrees and disagrees with the actual implementation. A box-and-arrow type diagram is used to depict the specified models and their differences. Their approach has helped with design reengineering and conformance tasks. This work is directed more toward static, architectural models, while our work is focused on more dynamic, protocol type models.

Due to the power and complexity of the OO paradigm, many efforts have developed to teach OO design and programming. While most of these efforts are lecture-driven courses, Robertson, et. al., have designed an interactive, scenario-based learning environment to teach OO principles[RCM+94]. Users of the system are guided through the OO design process by analyzing scenarios for a given problem domain. The goal is to develop an object model based on the dynamic scenarios. Scripts are created by the user to test the sequence of transactions between objects in their model, thereby simulating steps in the scenarios. Graphical views are provided which support the learning process. The authors intend to incorporate visualization support for education about the design patterns identified by Gamma, et. al.[GHJV95]. This work is addressing the dynamic understanding process during the design process, while ours is reverse engineering the design from the execution.

4 Impact and Future Work

The scenario described in this paper gives evidence of the conformance of automatically identified message patterns with design level abstractions. For this particular example, understanding the behavior of the `AnimateOne` pattern goes a long way toward understanding the entire execution–the various `AnimateOne` patterns constitute over 80 percent of the messages in the trace. While this will obviously not always be the case, it seems clear that message patterns can be used as abstractions to link the low-level implementation with higher level design abstractions during program understanding tasks.

These visualizations are only useful if they scale to handle real-world systems. We have discussed some techniques for storing and presenting large message traces, and are exploring different alternatives which vary the level of abstraction reflected by the message patterns. For example, we can ignore multiple iteration in the call trace or limit the stack depth, giving us "higher-level" message trace summaries which might be more useful for global understanding.

In addition to tracking message patterns, object instantiation and destruction patterns should be included, giving rise to more complete execution scenarios. This will require views which can abstract down to the object level. Another aspect of the current system that was not discussed are abstract source code views, which allow a user to relate the pattern information to the physical code with which (s)he actually works.

We are also adding several features to the visualizations to meet expected requirements for design validation and reengineering tasks. Rather than do the comparison of expected and actual behavior manually (as in the scenario), a facility which will allow a user to graphically input an interaction diagram and compare it with the various pattern views is being implemented. Essentially, there will be a way to "diff" patterns, graphically showing convergences and divergences. This will be useful for comparing similar patterns, such as two `AnimateOne` patterns, or a comparing user's diagram and an identified pattern.

Comparing similar patterns can lead to the identification of refactorings whereby similar behavior can be generalized. Comparing a user's model and implementation behavior (as in the usage

scenario) can be used to validate the implementation with respect to the design. A CASE tool with such a feature could help keep dynamic models up to date with respect to modifications of the implementation, a common problem in practice[MCL95]. Providing the capability for interatively creating interaction diagrams and other pattern views will also give the user a means for constructing a model of behavior for an unfamiliar design. Additionally, being able to visualize dynamic interactions is useful as a means for uncovering inefficient designs or implementations, and can be used help teach good OO design practices, as proposed by [RCM+94].

References

[AIS+77] Christopher Alexander, Sara Ishikawa, Murray Silverstein, Max Jacobson, and Ingrid Fiksdahl-King. *A Pattern Language.* Oxford University Press, New York, 1977.

[BH90] Heinz-Dieter Bocker and Jurgen Herczeg. What tracers are made of. In *Proceedings of the ECOOP/OOPSLA '90 Conference*, pages 89–99, Ottawa, Ontario, October 1990.

[CAB+94] D. Coleman, P. Arnold, S. Bodoff, C. Dollin, H. Gilchrist, F. Hayes, and P. Jeremaes. *Object-Oriented Development: The Fusion Method.* Prentice-Hall, Englewood Cliffs, NJ, 1994.

[CS95] James O. Coplien and Douglas C. Schmidt, editors. *Pattern Languages of Program Design.* Addison-Wesley, 1995.

[DBFPD95] S. Ducasse, M. Blay-Fornarino, and A. M. Pinna-Dery. A reflective model for first class dependencies. In *Proceedings of ACM OOPSLA '95*, pages 265–280, 1995.

[Dev92] P. Devanbu. A language and front-end independent code analyzer. In *Proceedings of the International Conference on Software Engineering*, Australia, May 1992.

[DPHKV93] Wim De Pauw, Richard Helm, Doug Kimelman, and John Vlissides. Visualizing the behavior of object-oriented systems. In *Proceedings of the ACM OOPSLA '93 Conference*, pages 326–37, Washington, D.C., October 1993.

[DPKV94] Wim De Pauw, Doug Kimelman, and John Vlissides. Modeling object-oriented program execution. In *Proceedings of the European Conference on Object-Oriented Programming '94*, 1994.

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Pattterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley, 1995.

[GKM83] S. L. Graham, P. B. Kessler, and M. K. McKusick. An execution profiler for modular programs. *Software Practice and Experience*, 13:671–685, 1983.

[Gol83] Adele Goldberg. *Smalltalk-80, The Interactive Programming Environment.* Addison-Wesley, Reading, PA, 1983.

[JS94] Dean F. Jerding and John T. Stasko. Using visualization to foster object-oriented program understanding. Technical Report GIT-GVU-94-33, Georgia Institute of Technology, July 1994.

[JS95a] Dean F. Jerding and John T. Stasko. The Information Mural: A technique for displaying and navigating large information spaces. In *Proceedings of the IEEE Visualization '95 Symposium on Information Visualization*, pages 43–50, Atlanta, GA, October 1995.

[JS95b] Dean F. Jerding and John T. Stasko. Using Information Murals in visualization applications. In *Proceedings of the 1995 Symposium on User Interface Software and Technology (Demonstration)*, pages 73–74, Pittsburgh, PA, November 1995.

[Kri94] Bent Bruun Kristensen. Complex associations: Abstractions in object-oriented modeling. In *Proceedings of ACM OOPSLA '94*, pages 272–283, Portland, OR, Oct 1994.

[LB92] James R. Larus and Thomas Ball. Rewriting executable files to measure program behavior. Technical Report Computer Sciences Technical Report 1083, University of Wisconsin-Madison, 1992.

[LC94] Panagiotis K. Linos and Vincent Courtois. A tool for understanding object-oriented program dependencies. In *Proceedings of the Workshop on Program Comprehension*, pages 20–27, Nov 1994.

[LM94] Chris Laffra and Ashok Malhotra. Hotwire – a visual debugger for C++. In *Proceedings of the USENIX 6th C++ Technical Conference*, April 1994.

[LN95a] Danny B. Lange and Yuichi Nakamura. Interactive visualization of design patterns can help in framework understanding. In *Proceedings of ACM OOPSLA '95*, pages 342–357, 1995.

[LN95b] Danny B. Lange and Yuichi Nakamura. Program Explorer: A program visualizer for C++. In *Proceedings of the USENIX Conference on Object-Oriented Technologies*, June 1995.

[MCL95] Ruth Malan, Derek Coleman, and Reed Letsinger. Lessons from the experiences of leading-edge object technology projects in Hewlett-Packard. In *Proceedings of ACM OOPSLA '95*, pages 33–46, 1995.

[MNS95] Gail C. Murphy, David Notkin, and Kevin Sullivan. Software Reflexion Models: Bridging the gap between source and high-level models. In *Proceedings of the Foundations of Software Engineering*, page ??, 1995.

[Moh88] Thomas G. Moher. PROVIDE: A process visualization and debugging environment. *IEEE Transactions on Software Engineering*, 14(6):849–57, June 1988.

[O'R88] Marign J. O'Riordan. Debugging and instrumentation of c++ programs. In *Proceedings of the USENIX C++ Conference*, pages 227–242, Denver, CO, Oct 1988.

[RBP+91] James Rumbaugh, Michael Blaha, William Premerlani, Frederick Eddy, and William Lorensen. *Object-Oriented Modeling and Design*. Prentice Hall, New York, NY, 1991.

[RCM+94] Scott P. Robertson, John M. Carroll, Robert L. Mack, Mary Beth Rosson, Sherman R. Alpert, and Jurgen Koenemann-Belliveau. ODE: A self-guided, scenario-based learning environment for object-oriented design principles. In *Proceedings of ACM OOPSLA '94*, pages 51–64, Portland, OR, Oct 1994.

[SK93] John T. Stasko and Eileen Kraemer. A methodology for building application-specific visualizations of parallel programs. *Journal of Parallel and Distributed Computing*, 18(2):258–264, June 1993.